Matt's Code

Matt James

Contents

1 Introduction **1**
- 1.1 Setting up the environment . . . 1
  - 1.1.1 Linux . . . 1
  - 1.1.2 Windows . . . 1
  - 1.1.3 MacOS . . . 2
- 1.2 Setting up a virtual environment . . . 2
- 1.3 Some python packages . . . 2

2 Plasma Models **5**
- 2.1 `spicedmodel`: The Scalable Plasma Ion Composition and Electron Density Model . . . 5
- 2.2 `spiced` . . . 5
  - 2.2.1 Installation . . . 5
  - 2.2.2 Usage . . . 5
- 2.3 `HermeanFLRModel`: Model of Mercury's dayside plasma mass density . . . 5
  - 2.3.1 Installation . . . 5
  - 2.3.2 Usage . . . 6
- 2.4 `PyGCPM`: Wrapper for the Global Core Plasma Model . . . 6
  - 2.4.1 Installation . . . 6
  - 2.4.2 Usage . . . 6

3 Field Models **9**
- 3.1 `PyGeopack`: Python wrapper for the Tsyganenko field models . . . 9
  - 3.1.1 Installation . . . 9
  - 3.1.2 Usage . . . 9
- 3.2 `geopack`: C++ wrapper for Tsyganenko field models . . . 10
  - 3.2.1 Installation . . . 11
  - 3.2.2 Usage . . . 11
- 3.3 `libinternalfield`: C++ spherical harmonic model code . . . 11
  - 3.3.1 Installation . . . 11
  - 3.3.2 Usage . . . 12
- 3.4 `vsmodel`: Python based Volland-Stern electric field model for Earth . . . 12
  - 3.4.1 Installation . . . 12
  - 3.4.2 Usage . . . 13
- 3.5 `JupiterMag`: Python wrapper for Jovian field models . . . 15
- 3.6 `libjupitermag`: C++ library for field tracing in Jupiter's magnetosphere . . . 15
- 3.7 `con2020`: Python implementation of Jupiter's magnetodisc model . . . 15
- 3.8 `libcon2020`: C++ implementation of Jupiter's magnetodisc model . . . 15
- 3.9 `jrm33`: The JRM33 model in Python . . . 15
- 3.10 `jrm09`: The JRM09 model in Python . . . 15
- 3.11 `vip4model`: The VIP4 model in Python . . . 15

4 Spacecraft Data **17**
- 4.1 `Arase`: Download and read Arase data . . . 17
- 4.2 `RBSP`: Download and read Van Allen Probe data . . . 17
- 4.3 `cluster`: Download and read Cluster data . . . 17
- 4.4 `pyCRRES`: Download and read CRRES data . . . 17
- 4.5 `themissc`: Download and read THEMIS data . . . 17
- 4.6 `imageeuv`: Download and read IMAGE EUV data . . . 17
- 4.7 `imagerpi`: Download and read IMAGE RPI data . . . 17

4.8 imagePP: Download and read Goldstein's plasmapause dataset . . . 17
4.9 PyMess: Download and read MESSENGER data . . . 17
4.10 FIPSProtonData: Download and read ANN verified FIPS moments . . . 17
4.11 VenusExpress: Download and read VEX data . . . 17

5 Ground Data and Geomagnetic Indices **19**
5.1 groundmag: Tools for processing and reading ground magnetometer data . . . 19
5.2 SuperDARN: Simple SuperDARN fitacf reading code . . . 19
5.2.1 Installation . . . 19
5.3 kpindex: Download the latext Kp indices . . . 21
5.4 pyomnidata: Download the latext OMNI and solar flux data . . . 21
5.5 smindex: Read the SuperMAG indices . . . 21

6 Machine Learning **25**
6.1 NNClass: Simple neural network classifier module . . . 25
6.2 NNFunction: Train neural networks on arbitrary functions . . . 25

7 Other Tools **27**
7.1 wavespec: Spectral analysis tools . . . 27
7.2 MHDWaveHarmonics: Tools for MHD waves . . . 27
7.3 FieldTracing: Python field tracing code . . . 27
7.4 DateTimeTools: Tools for dealing with dates and times . . . 27
7.5 datetime: C++ library dealing for dates and times . . . 27
7.6 PyFileIO: Tools for reading and writing files . . . 27
7.7 RecarrayTools: Tools for manipulating numpy.recarrays . . . 27
7.8 PBSJobExamples: Examples for submitting jobs to PBS . . . 27
7.9 PlanetSpice: SPICE related code . . . 27
7.10 ColorString: Change colour of strings in the terminal . . . 27
7.11 cppembedbinary: Examples for embedding data into C++ code . . . 27
7.12 libspline: C++ library for splines . . . 27
7.13 linterp: C++ interpolation code . . . 27

Chapter 1

Introduction

This document lists a bunch of the GitHub repositories created by me which may be useful to others. Some of these repositories are fairly complete, others are less so. I will do my best to fix and update anything that is buggy or incomplete, please do report bugs in the relevant repositories if you can. If you're feeling particularly helpful - feel free to send pull requests.

Most of the code here is written in Python, some things make use of C++ libraries to do some of the heavy lifting, one is a pretty dodgy Python wrapper of a C++ wrapper of Fortran code... Some of the modules and libraries used here are dependencies of others. In the more complete repos `pip` will take care of dependencies, otherwise some manual installation may be required.

1.1 Setting up the environment

In this section I describe how to set up the environment such that everything *should* pretty much work...

1.1.1 Linux

If running on ALICE/SPECTRE, you will most likely be required to enable the following modules:

```
module load gcc/9.3
module load python/gcc/3.9.10
module load git/2.35.2
```

where exact version numbers may change (use whatever is latest, don't just copy and paste!) and the replacement for SPECTRE/ALICE may have another method for loading these things in for all I know. I also recommend adding those to the end of your `/.bashrc` file so that they load every login, e.g.:

```
echo module load gcc/9.3 >> ~/.bashrc
echo module load python/gcc/3.9.10 >> ~/.bashrc
echo module load git/2.35.2 >> ~/.bashrc
```

The above is unlikely to be necessary on a local Linux installation, instead I would recommend installing `git, gcc, g++, make, gfortran` and `pip3`, e.g. in Ubuntu:

```
sudo apt install git gcc g++ binutils gfortran python3-pip
```

All of the above should allow you to install/run/compile most of my code. I wouldn't recommend using Conda in Linux - I know it has cause some problems/confusion when it comes to linking Python with C/C++ on SPECTRE.

1.1.2 Windows

A fair portion of the code is able to run on Windows - much of the Python code is platform independent and some of the C++ libraries/backends are able to be compiled using Windows. In this case, I *would* actually recommend installing Conda, as it worked for me. The GCC compilers (for C/C++/Fortran) can all be installed easily with TDM-GCC (get the 64-bit version here), just remember to put a tick in the box for "fortran" and "openmp".

1.1.3 MacOS

I managed to install the relevant packages in a virtual Hackintosh once. I don't remember how, perhaps using homebrew. Good luck...

1.2 Setting up a virtual environment

In SPECTRE I never actually bothered with a virtual environment, mistakes were made, headaches may have been avoided had I done so. This step is entirely optional, but somewhat recommended:

```
#create a virtual environment, call it what you want,
#here I call mine "env"
python3 -m venv env
```

Once this has been created, you MUST activate it before running any code, or you will just be running things globally:

```
source env/bin/activate
```

note that I am assuming that `env` exists in the current working directory, if not adjust the path accordingly! If it works, the prompt terminal prmpt should change, e.g:

```
#before:
matt@matt-MS-7B86:~$

source env/bin/activate
#after:
(env) matt@matt-MS-7B86:~$
```

1.3 Some python packages

Here are a list of Python packages which are either going to be required by most of my code, or would just be recommended:

1. ipython : best Python interpreter, forget notebooks
2. numpy : essential, don't skip
3. matplotlib : for plotting
4. scipy : loads of good stuff here
5. wheel : used to build Python packages to be installed by pip
6. cdflib : reads CDF files
7. keras : nice for machine learning
8. tensorflow : also machine learning

install them:

```
#update pip first
python3 -m install pip --upgrade --user

pip3 install ipython numpy matplotlib scipy wheel cdflib keras tensorflow --user
```

where the "`--user`" flag may or may not be necessary, depending on your version of Python - it places the installed modules in `~/.local/lib/python3.9/site-packages`.

In theory, at this point you should be able to run `ipython3` (or just `ipython`) within the terminal, from which any installed code can be imported. The reason I recommend using Ipython over the standard Python interpreter is that it has autocomplete and it uses pretty colours for syntax highlighting. It would also be a good idea to enable the autoreload feature in Ipython, which recompiles anything that has been edited since it was last run, otherwise would have to reload the code manually (or restart the session) after every edit. Run

```
ipython profile create
```

then add the following lines to ∼/.ipython/profile_default/ipython_config.py:

```
c.InteractiveShellApp.extensions = ['autoreload']
c.InteractiveShellApp.exec_lines = ['%autoreload 2']
c.InteractiveShellApp.exec_lines.append('print("Warning: disable autoreload in ipython_config.py to imp
```

That should just about do it.

Chapter 2

Plasma Models

2.1 `spicedmodel`: The Scalable Plasma Ion Composition and Electron Density Model

2.2 `spiced`

GitHub: https://github.com/mattkjames7/spiced.git

The C++ code behind the SPICED model. It should be possible to build this library in Linux, Windows and MacOS.

2.2.1 Installation

In Linux and MacOS, it should eb possible to make and install the library:

```
make

#optionally install globally
sudo make install
```

Or in Windows:

```
compile.bat
```

2.2.2 Usage

When using this library, the header file should be included, i.e.:

```
#include <spiced.h>
```

and the linker flag `-lspiced` should be used during compilation.

The models also need to be initialized at runtime using `initModels();`. `spiced.h` contains a full list of the functions which can be linked to using C and other languages like Python within the `extern "C" {}` section; other symbols outside this section can, such as the model objects themselves may be interacted with directly using C++.

2.3 `HermeanFLRModel`: Model of Mercury's dayside plasma mass density

GitHub: https://github.com/mattkjames7/HermeanFLRModel.git

Estimate the dayside plasma mass density in Mercury's magnetosphere using the field line resonance (FLR) based model from James2019.

2.3.1 Installation

This hasn't been placed on PyPI, so either download fromt he GitHub page and install using `pip`, or clone and build the package:

```
#if you download the package
pip3 install HermeanFLRModel-x.y.z-py3-none-any.whl --user

#or clone, build and install
git clone https://github.com/mattkjames7/HermeanFLRModel.git
cd HermeanFLRModel
python3 setup.py bdist_wheel
pip3 install dist/HermeanFLRModel-x.y.z-py3-none-any.whl --user
```

where `x.y.z` should be replaced with the current version number.

2.3.2 Usage

Import the module and create an instance of the `Model` object:

```
import HermeanFLRModel as hflr

model = hflr.Model(Alpha,Coord='MSM')
```

where `Alpha` is the power law index which should be an integer from 0 to 6, and `Coord` sets the coordinate system to use (either MSM or MSO).

Use the `model.Calc()` member function to work out densities:

```
#create input coordinate(s)
x = np.zeros(6)
y = np.array([1.0,1.2,1.4,1.6,1.8,2.0])
z = np.zeros(6)

#call model Calc() function
rho = model.Calc(x,y,z)
```

Or produce a plot of the plasma mass denity for a slice through the magnetosphere, e.g.:

```
#select magnetic local time and alpha
MLT = 6.0
Alpha = 3.0

#plot it
ax = hflr.PlotModelSlice(MLT,Alpha)
```

which should produce something like figure 2.1.

2.4 PyGCPM: Wrapper for the Global Core Plasma Model

GitHub: https://github.com/mattkjames7/PyGCPM.git

This is a Python wrapper for the Global Core Plasma Model (Gallagher2000, code found here)

2.4.1 Installation

This module exists on PyPI, so can be installed using `pip`:

```
pip3 install PyGCPM --user
```

2.4.2 Usage

There are three functions:

1. `PyGCPM.GCPM()`: provides particle densities at positions defined in SM coordinates.
2. `PyGCPM.PlotEqSlice()` : plots the density of a species in the equatorial plane.
3. `PyGCPM.PlotMLTSlice()` : plots the density of a particle species in

Firstly, get some densities at some positions in SM coordinates, units of R_E:

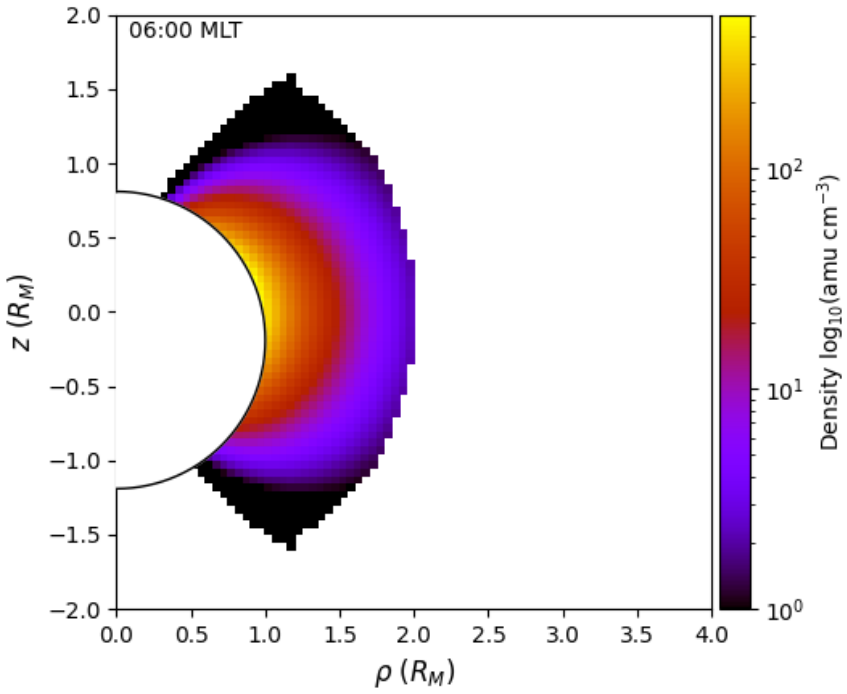

Figure 2.1: Plasma mass density at 6:00 MLT.

```
import PyGCPM
ne,nH,nHe,nO = PyGCPM.GCPM(x,y,z,Date,ut,Kp=Kp,Verbose=Verbose)
```

where `Date` is the date in the format yyyymmdd, `ut` is the time in hours, `Kp` is the Kp index and `Verbose=True` would display progress. The outputs of this function `ne`, `nH`, `nHe` and `nO` are the densities of electrons, protons, helium ions and oxygen ions, respectively.

We can plot the density of a particle species in the equatorial plane:

```
PyGCPM.PlotEqSlice(20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.2.

We can also plot the density of a particle species in a slice of MLT:

```
PyGCPM.PlotMLTSlice(8.0,20010902,12.0,Parameter='ne')
```

which should produce the plot in figure 2.3.

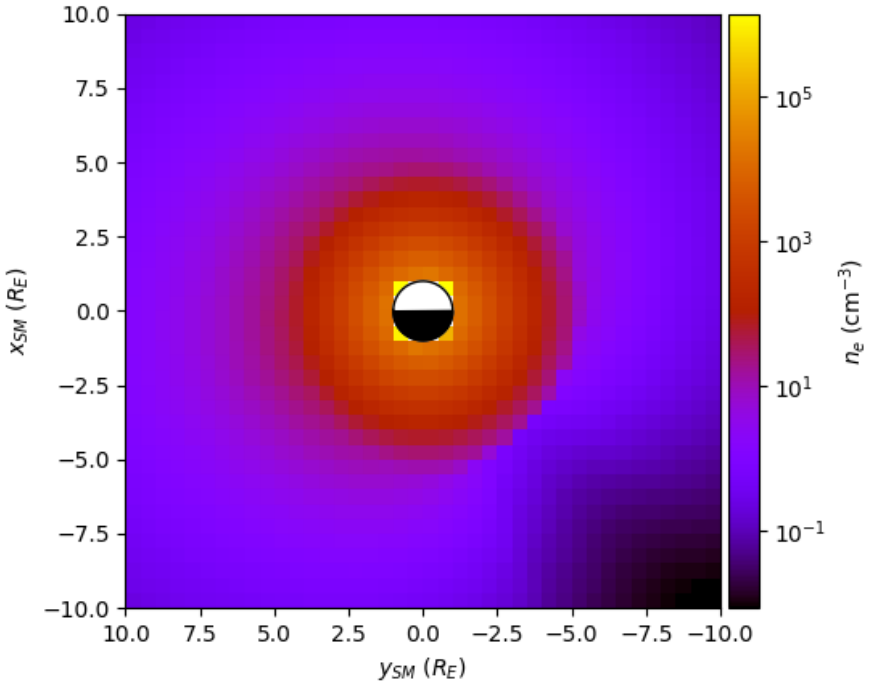

Figure 2.2: Equatorial electon density.

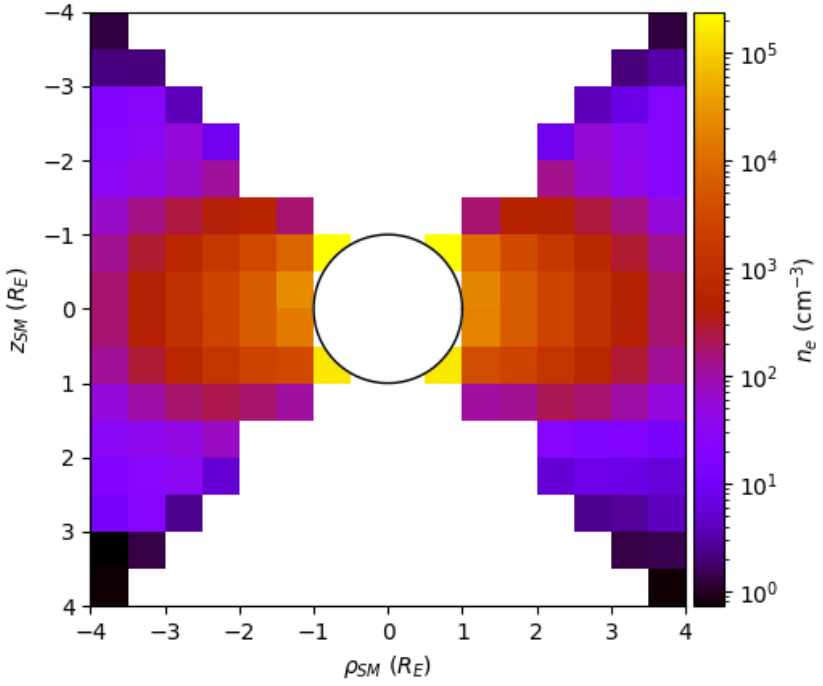

Figure 2.3: MLT slice of electon density.

Chapter 3

Field Models

3.1 `PyGeopack`: Python wrapper for the Tsyganenko field models

This is a Python module for obtaining field vectors and traces of the Tsygenenko field models. It is a wrapper of a wrapper (see 3.2). The latest code can be viewed and downloaded from here: https://github.com/mattkjames7/PyGeopack.

3.1.1 Installation

The easiest way to install `PyGeopack` is using `pip`, e.g.:

```
pip3 install PyGeopack --user
```

for other installation methods, see the GitHub repo.

At this point, it *may* just work if you were to try to import it, but there's a reasonably good chance that the C++/Fortran code will need to be recompiled, in which case we need to ensure that there are compilers available to do this (see section sectSetup).

One of the features of PyGeopack is that it can easily package together all of the geomagnetic/solar wind parameters that the models use so that when you request a trace or a field vector for a specific date and time, it will automatically try to find the appropriate parameters. This isn't strictly necessary for the models to work, as they will default to some fairly average parameters and they can be overridden manually. In order to be able to use this functionality, this module and the submodules which it relies on to collect the relevant data need to know where they can store the parameters. This means exporting a few environment variables (e.g. in ~`/.bashrc`):

```
export KPDATA_PATH=/path/to/kp
export OMNIDATA_PATH=/path/to/omni
export GEOPACK_PATH=/path/to/geopack/data
```

which are set as follows for me on SPECTRE:

```
export KPDATA_PATH="/data/sol-ionosphere/mkj13/Kp"
export OMNIDATA_PATH="/data/sol-ionosphere/mkj13/OMNI"
export GEOPACK_PATH="/data/sol-ionosphere/mkj13/Geopack"
```

Once that is done, it should work...

3.1.2 Usage

The first time this is imported, there is a good chance that it will attempt to recompile itself. There will be a lot of messages on the screen, but it should finish successfully. If it fails, double check that you have the required compilers installed, raise an issue on the GitHub page if the problem persists.

If you would lke the latest model parameters, run the following:

```
import PyGeopack as gp
gp.Params.UpdateParameters(SkipWParameters=True)
```

This may take a little time, depending on how much data it needs to download. It will take all of the data and compile it into one binary file ~350 MiB in size, once this is done, it should be relatively quick loading the data into memory.

The model field vectors can be returned using the `ModelField` function:

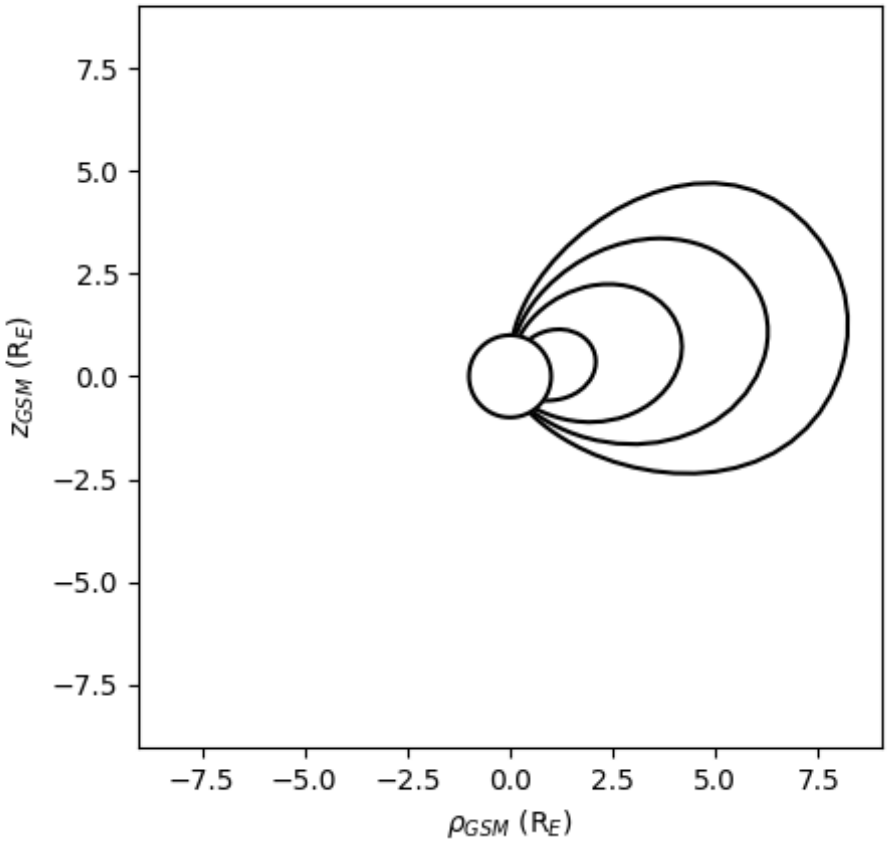

Figure 3.1: Example of field tracing using `PyGeopack`.

```
Bx,By,Bz = gp.ModelField(x,y,z,Date,ut,Model='T96',CoordIn='GSM',**kwargs)
```

where `x`, `y` and `z` can be scalars or arrays of position, in units of Earth radii and in the coordinate system defined by the `CoordIn` keyword (`'GSE'|'GSM'|'SM'`). `Date` can be an array or scalar of date(s) in the format yyyymmdd, while `ut` is in hours from the start of the day. The models currently available are `'T89'`, `'T96'`, `'T01'` and `'TS05'`.

Traces are simple to produce and can be done a single trace at a time, or in batches, e.g.:

```
import numpy as np

#define a few starting positions for the traces
x = np.array([2.0,4.0,6.0,8.0])
y = np.array([0.0,0.0,0.0,0.0])
z = np.array([0.0,0.0,0.0,0.0])

#run the traces, return TraceField object
T = gp.TraceField(x,y,z,20221222,16.0)

#plot field traces
ax = T.PlotRhoZ()
```

which should produce a plot similar to figure 3.1.

For more information on the keyword arguments please see the readme.

3.2 geopack: C++ wrapper for Tsyganenko field models

This is the code that `PyGeopack` calls, it provides a simple C-compatible interface for calculating field vectors, tracing and coordinate conversion. The C++ code in this library is used for configuring model parameters,

determining footprints and providing a simple interface for the models, while the Fortran code currently provides field vectors, coordinate transforms and tracing.

3.2.1 Installation

This code can be installed as a linkable library (at least on POSIX systems):

```
#clone the repo
git clone https://github.com/mattkjames7/geopack --recurse-submodules

#cd into it
cd geopack

#fetch the submodules if you forgot the
#--recurse-submodules flag on the clone command
git submodule update --init --recursive

#compile it
make
sudo make install
```

On Linux and MacOS the `sudo make install` command will place the header file at `/usr/local/include/geopack.h` and the shared object file at `/usr/local/lib/libgeopack.so`, unless the `PREFIX` keyword is set (by default `PREFIX=/usr/local`). If the `PREFIX` uses a custom path, then be sure to let the linker know where it exists, either by setting `LD_LIBRARY_PATH` or by using `-L` and `-I`.

On Windows, run `compile.bat` to create `lib
libgeopack.dll`.

3.2.2 Usage

To use this code with C and C++, the header file must be included, i.e.:

```
#include <geopack.h>
```

and when compiling, the `-lgeopack` flag should be used to link to the library.

C and other languages such as Python should use the wrapper functions defined within the `extern "C" {}` section of `geopack.h`. This includes `ModelField()` for calculating model field vectors, `TraceField()` for field traces and a number of functions for coordinate conversion, e.g. `SMtoGSMUT()`. An example of how to trace a field line is in `geopack.c`.

C++ is able to use all of the functions declared in `geopack.h`, including the wrapper functions used by C. This means that direct usage of the `Trace` class is possible, an example of this is shown in `geopack.cc`.

3.3 `libinternalfield`: C++ spherical harmonic model code

This library provides field vectors for spherical harmonic magnetic field models. It has a bunch of built in models from magnetized planets and a moon (Ganymede). This library is used by libjupitermag.

3.3.1 Installation

Compile and install in Linux and MacOS:

```
#clone the repo
git clone https://github.com/mattkjames7/libinternalfield

#cd into it
cd libjupitermag

#compile it
make
sudo make install
```

Or in Windows, run `compile.bat` to create `libinternalfield.dll`.

3.3.2 Usage

To use this library, the header should be included `include <internalfield.h>` and the code should be compiled with the `-linternalfield` flag in order to link to the library.

In C, we can use the `getModelFieldPointer()` to get a function pointer to the model we want to use, e.g.:

```c
#include <stdio.h>
#include <internalfield.h>

int main() {

        printf("Testing C\n");

        /* try getting a model function */
        modelFieldPtr model = getModelFieldPtr("jrm33");
        double x = 10.0;
        double y = 10.0;
        double z = 0.0;
        double Bx, By, Bz;
        model(x,y,z,&Bx,&By,&Bz);

        printf("B = [%6.1f,%6.1f,%6.1f] nT at [%4.1f,%4.1f,%4.1f]\n",Bx,By,Bz,x,y,z);

        printf("C test done\n");

}
```

C++ can use the `internalModel` object, e.g.:

```c
#include <internal.h>

int main() {
    /* set current model */
    internalModel.SetModel("jrm09");

    /* set intput and output coordinates to Cartesian */
    internalModel.SetCartIn(true);
    internalModel.SetCartOut(true);

    /* input position (cartesian)*/
    double x = 35.0;
    double y = 10.0;
    double z = -4.0;

    /* output field */
    double Bx, By, Bz;
    internalModel.Field(x,y,z,&Bx,&By,&Bz);
}
```

3.4 `vsmodel`: Python based Volland-Stern electric field model for Earth

This is a purely Python-based implementation of the Volland-Stern electric field model Volland1973,Stern1975.

3.4.1 Installation

Use `pip`:

```
pip3 install vsmodel --user
```

3.4.2 Usage

Electric field vectors can be determined using either cylindrical (`vsmodel.ModelE`) or Cartesian (`vsmodel.ModelCart`) coordinates (Solar Magnetic), e.g.:

```
import vsmodel

##### The simple model using Maynard and Chen ####
#the Cartesian model
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp)


#### The Goldstein et al 2005 version ####
#the Cartesian model, either by providing solar wind
#speed (Vsw) and IMF Bz (Bz), or the equivalent E field (Esw)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Vsw=Vsw,Bz=Bz)
Ex,Ey,Ez = vsmodel.ModelCart(x,y,Kp,Esw=Esw)

#the cylindrical model
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Vsw=Vsw,Bz=Bz)
Er,Ep,Ez = vsmodel.ModelE(r,phi,Kp,Esw=Esw)
```

where the Maynard1975 method uses the Kp index and the Goldstein2005 method uses Kp, solar wind speed and the z-component of the interplanetary magnetic field.

The electric field magnitude and $\mathbf{E} \times \mathbf{B}$ velocity can be plotted in the Earth's equatorial plane:

```
import vsmodel
import matplotlib.pyplot as plt

plt.figure(figsize=(9,8))
ax0 = vsmodel.PlotModelEq('E',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax1 = vsmodel.PlotModelEq('E',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,0],fig=plt,fmt='%4.2f',scale=[0.01,10.0])
ax2 = vsmodel.PlotModelEq('V',Kp=1.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,0,1],fig=plt,scale=[100.0,10000.0])
ax3 = vsmodel.PlotModelEq('V',Kp=5.0,Vsw=-400.0,Bz=-2.5,
                maps=[2,2,1,1],fig=plt,scale=[100.0,10000.0])
ax0.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax2.set_title('$K_p=1$; $E_{sw}=-1$ mV m$^{-1}$')
ax3.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
ax1.set_title('$K_p=5$; $E_{sw}=-1$ mV m$^{-1}$')
plt.tight_layout()
```

which would produce figure 3.2

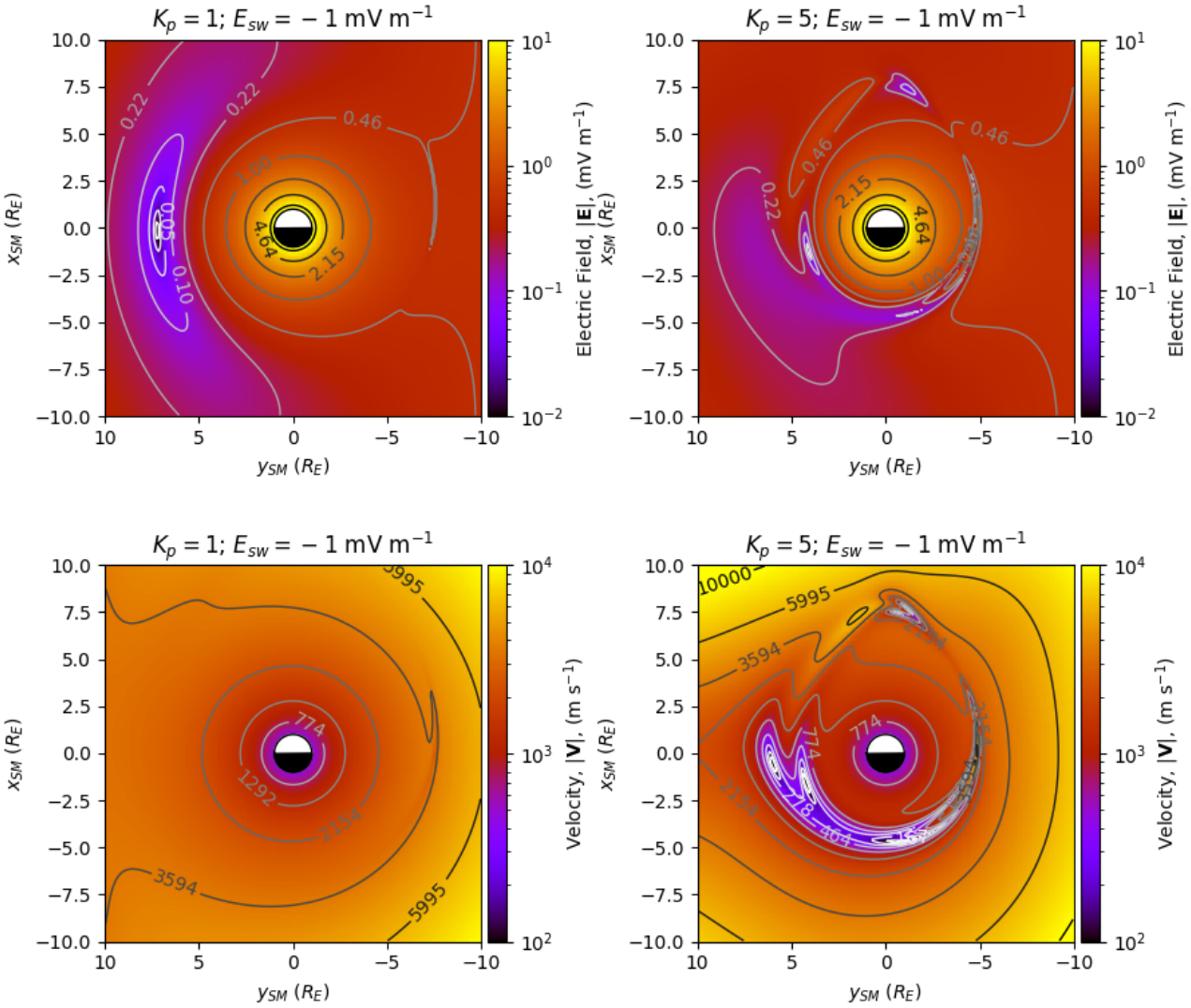

Figure 3.2: Volland-Stern electric field (top plots) and $\mathbf{E} \times \mathbf{B}$ velocity (bottom plots).

3.5 JupiterMag: Python wrapper for Jovian field models

3.6 libjupitermag: C++ library for field tracing in Jupiter's magnetosphere

3.7 con2020: Python implementation of Jupiter's magnetodisc model

3.8 libcon2020: C++ implementation of Jupiter's magnetodisc model

3.9 jrm33: The JRM33 model in Python

3.10 jrm09: The JRM09 model in Python

3.11 vip4model: The VIP4 model in Python

Chapter 4

Spacecraft Data

4.1 `Arase`: Download and read Arase data

4.2 `RBSP`: Download and read Van Allen Probe data

4.3 `cluster`: Download and read Cluster data

4.4 `pyCRRES`: Download and read CRRES data

4.5 `themissc`: Download and read THEMIS data

4.6 `imageeuv`: Download and read IMAGE EUV data

4.7 `imagerpi`: Download and read IMAGE RPI data

4.8 `imagePP`: Download and read Goldstein's plasmapause dataset

4.9 `PyMess`: Download and read MESSENGER data

4.10 `FIPSProtonData`: Download and read ANN verified FIPS moments

4.11 `VenusExpress`: Download and read VEX data

Chapter 5

Ground Data and Geomagnetic Indices

5.1 groundmag: Tools for processing and reading ground magnetometer data

5.2 SuperDARN: Simple SuperDARN fitacf reading code

https://github.com/mattkjames7/SuperDARN

The `SuperDARN` module is for reading and plotting SuperDARN fitacf files. It is a fairly simple tool, but use with caution because there may be some errors...

5.2.1 Installation

This package is not in the PyPI, so manual installation is necessary:

```
#clone the repo
git clone https://github.com/mattkjames7/SuperDARN
cd SuperDARN

#build a Python package
python3 setup.py bdist_wheel

#install it (replace 0.1.0 with whatever version is built)
pip3 install dist/SuperDARN-0.1.0-py3-none-any.whl --user
```

Once installed, the directory used to create the Python wheel file can be deleted. It can be uninstalled using `pip3 uninstall SuperDARN`.

Before running for the first time, a couple of environment variables need to be set up to tell the module where to look for fitacf files and to say where it is able to store some files:

```
#path to where FITACF files are stored
#(this one is specific to SPECTRE)
export FITACF_PATH=/data/sol-ionosphere/fitacf

#path to where this module can create some files
#(this should be a path where you have write access)
export SUPERDARN_PATH=/some/other/path/SuperDARN
```

This module will not currently run on Windows (as far as I am aware) because it requires the compilation of some C++ code which is not yet cross-platform.

Usage

In ipython, the first time this module is imported, it should attempt to download some files from the Radar Software Toolkit (RST) which help in calculating the coordinates of the fields of view of each radar. These files are created in the path defined be the `$SUPERDARN_PATH` variable.

Reading Data

There are a few functions within `SuperDARN.Data` which provide objects containing data:

```
import SuperDARN as sd

#get the data from a single cell (Radar,Date,ut,Beam,Gate)
cdata = sd.Data.GetCellData('han',20020321,[22.0,24.0],9,25)

#or a whole beam of data (Radar,Date,ut,Beam)
bdata = sd.Data.GetBeamData('han',[20020321,20020322],[22.0,24.0],7)

#data for the whole field of view (Radar,Date,ut)
#in this case, the output is a dict where each key is a beam number
#pointing to a recarray for each beam as produced by GetBeamData
rdata = sd.Data.GetRadarData('han',[20020321,20020322],[22.0,23.0])
```

In the above examples `bdata` and `cdata` are `numpy.recarray` objects, `rdata` is a `dict` object containing a `numpy.recarray` for each beam.

The fitacf data are stored in memory once loaded so that they don't need to be re-read every time the data are requested. To check how much memory is in use and to clear it:

```
#check memory usage in MB
sd.Data.MemUsage()

#clear memory
sd.Data.ClearData()
```

Plotting Data

There are a bunch of very simple plotting functions, e.g.:

```
import matplotlib.pyplot as plt

#create a figure
plt.figure(figsize=(8,11))

#plot the power along a beam
ax0 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],
                                        Param='P_l',ShowScatter=True,fig=plt,
                                        maps=[2,3,0,0],scale=[1.0,100.0],zlog=True,
                                        cmap='gnuplot')

#the velocity
ax1 = sd.Plot.RTIBeam('han',[20020321,20020322],[23.0,1.0],9,[20,35],Param='V',
                                        fig=plt,maps=[2,3,1,0])

#velocity along a range of latitudes at a ~constant longitude of 105
ax2 = sd.Plot.RTILat('han',[20020321,20020322],[23.0,1.0],105.0,Param='V',
                                        fig=plt,maps=[2,3,0,1])

#velocity along a range of longitudes at a ~constant latitude of ~70
ax3 = sd.Plot.RTILon('han',[20020321,20020322],[23.0,1.0],70.0,Param='V',
                                        fig=plt,maps=[2,3,1,1])

#some specific cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
ax4 = sd.Plot.RTI('han',[20020321,20020322],[23.0,1.0],beams,gates,
                                        Param='V',fig=plt,maps=[2,3,0,2])

#totally different FOV plot
```

```
ax5 = sd.Plot.FOVData('han',20020321,23.5,Param='V',fig=plt,maps=[2,3,1,2])


plt.tight_layout()
```

which should produce figure 5.1.

Fields of View

These may be wrong. Use with great caution.

The fields of view of each radar are stored as instances of the `SuperDARN.FOV.FOVObj` objects in memory and can be accessed using `GetFOV`, e.g.:

```
#get the object from memory
Date = 20020321
fov = sd.FOV.GetFOV('pyk',Date)

#use it to retrieve the FOV in mag coordinates
mlon,mlat = fov.GetFOV(Mag=True,Date=Date)

#plot it
ax = fov.PlotPolar(Background=[0.0,0.2,1.0],Continents=[0.0,1.0,0.2],
                                color='magenta',ShowBeams=False,ShowCells=False,
                                linewidth=2.0,Mag=True,Lon=True)

#add some cells
beams = [1,5,7,2,8,4,9]
gates = [20,26,33,22,25,21,29]
fov.PlotPolarCells(beams,gates,color='red',fig=ax,Mag=True,linewidth=2.0,Lon=True)
```

The above code should look like figure 5.2:

5.3 kpindex: Download the latext Kp indices

5.4 pyomnidata: Download the latext OMNI and solar flux data

5.5 smindex: Read the SuperMAG indices

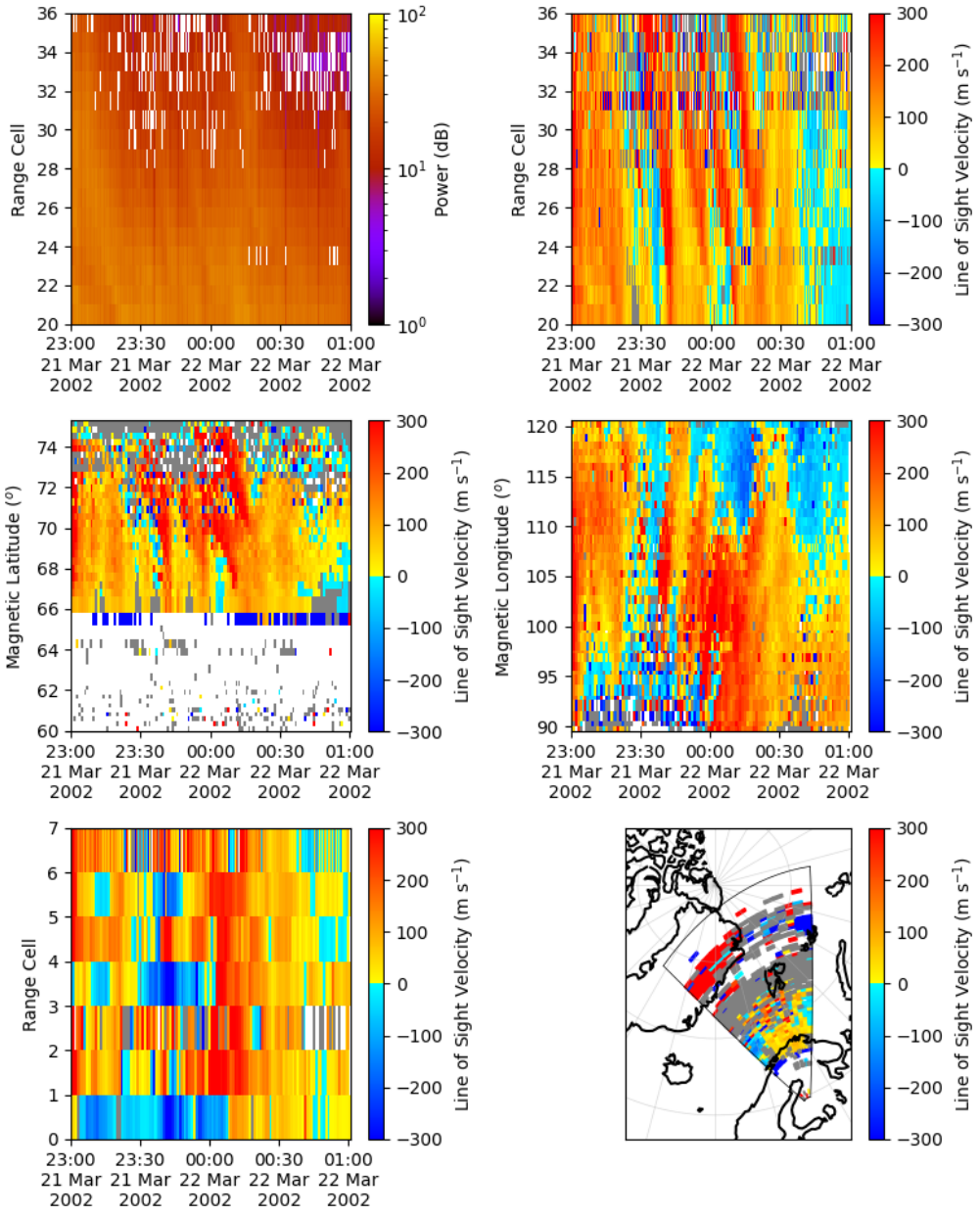

Figure 5.1: Top left: range time intensity (RTI) plot of backscatter power. Top right: RTI plot of line of sight velocity. Mid left: velocity along a line of cells in magnetic longitude. Mid right: velocity along a range of longitudes. Bottom left: velocity of specific range cells. Bottom right: velocity within the field of view plot.

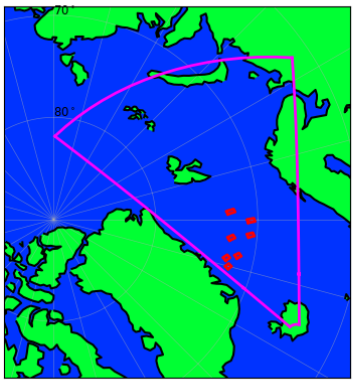

Figure 5.2: SuperDARN field of view plot with specific cells highlighted.

Chapter 6

Machine Learning

6.1 `NNClass`: Simple neural network classifier module

6.2 `NNFunction`: Train neural networks on arbitrary functions

Chapter 7

Other Tools

7.1 `wavespec`**: Spectral analysis tools**

7.2 `MHDWaveHarmonics`**: Tools for MHD waves**

7.3 `FieldTracing`**: Python field tracing code**

7.4 `DateTimeTools`**: Tools for dealing with dates and times**

7.5 `datetime`**: C++ library dealing for dates and times**

7.6 `PyFileIO`**: Tools for reading and writing files**

7.7 `RecarrayTools`**: Tools for manipulating** `numpy.recarrays`

7.8 `PBSJobExamples`**: Examples for submitting jobs to PBS**

7.9 `PlanetSpice`**: SPICE related code**

7.10 `ColorString`**: Change colour of strings in the terminal**

7.11 `cppembedbinary`**: Examples for embedding data into C++ code**

7.12 `libspline`**: C++ library for splines**

7.13 `linterp`**: C++ interpolation code**